Optimizing Sequential Key Inserts into B-Tree

Abstract

B-Trees are one of the most widely used data structures in database systems, used to index data from a variety of sources. As servers running RDBMS's have increased the number of processors available, the demand for data structures that can support concurrent operations has increased along with it. Many extensions of B-Trees have been proposed that minimize locking (e.g. Latch Coupling [4], B^{link}-Link Trees [5], Optimistic Lock Coupling [1]) or eliminate locking completely (e.g. Bw-Trees [2]) to improve concurrency. While these methods have greatly increased to concurrent throughput over baseline locking algorithms, concurrent sequential key inserts are still problematic for many algorithms, and lead to greatly reduced throughput compared to other workloads. In this paper we examine the problem of sequential key inserts and propose a solution that extends a B-Tree Optimistic Lock Coupling to reduce contention between threads and improve throughput of this typically challenging workload.

Background

Related Work

Bw-Trees

B-Tree with OLC

Other Tree Structures

Problem

Consider the following workflow, 1. A customer submits an order 2. The server adds additional information about the order and gets the next unique order id (monotonically increasing) 3. The server then insert the completed order into the database 4. The server then notifies the customer that the order is placed

In this scenario it is likely that the order ids will be indexed as they make an ideal primary key. For a hash index, such a workflow is not an issue since the hashing should randomly distribute the keys and hence contention between concurrent modifications should be low. For a B-Tree however this is very problematic. Because the order ids are monotonically increasing, all concurrent threads are going to be inserting into the right most leaf of the tree. For algorithms that take locks to perform updates, this is going to hurt performance since all threads need to take the same lock to insert the order. It is of course possible to randomly select order ids, turning the sequential inserts into a random inserts, however you loose the ability to quickly iterate over the orders in chronological order. For similar reasons a hash index might not be appropriate. Iterating over orders in chronological order maybe very useful when determining which orders to fill first in the event of limited supply hence maintaining a monotonically increasing order id in a B-Tree index adds functionality that is not present in other solutions.

Solution

As illustrated above, scenarios certain workloads are very problematic for current B-Tree implementations. The key observation is that most algorithms breakdown when there are many concurrent writers on the same leaf of the tree, due to locking for writes. The main goal of our solution is to get rid of locking for writes that would normally be high contention. To do this we propose a two step solution when handling inserts into the B-Tree. When inserting, we first put the insert into an unordered buffer. Once the buffer fills, the buffer is replaced with another buffer from a preallocated pool and a thread is assigned to flush the buffer into the tree using repeated inserts into the B-Tree. We choose to use a B-Tree with Optimistic Lock Coupling since [3] showed that of the B-Tree, it performed the best of the B-Trees tested.

Insert Buffer

Our buffer uses implementation uses a single lock to synchronize before flushing to the B-Tree, an atomic integer for the next insert postition, an atomic integer for the minimum version number, and a fixed size array (current 1024 elements) to store the inserts. Before a thread is allow to try to insert into the buffer, it must first acquire a *shared* lock on the buffer before trying to insert into the buffer. Once the a shared lock on the buffer is acquired, the thread doesn't realease it until the buffer is filled. After the acquring a shared lock on the buffer, the thread then attempts to insert into the buffer by atomicly fetching and incrementing the current insert position, if the position is valid, the thread inserts the key, value pair into that position in the buffer. If the position is not valid, the insert fails and the failure is returned to the thread. Note that while the getting the insert position is atomic, writing to the buffer is not. Because of the lock synchronization this is not an issue when the buffer is flushed but could potenially be an issue when the buffer read, we address this issue below.

```
struct InsertBuffer {
    static constexpr long capacity = 1024;
    std::shared_mutex mu;
    std::atomic<long> pos, min_version;
    std::array< std::pair<K, Versioned<V> >, capacity> buf;
}
```

Ordering of Inserts

While unordered buffers make concurrent inserts easy, it can lead to unintuitive behavior. Say we have a thread *t1* performing inserts into a *unique* B-Tree. *t1* inserts pair (*k*, *v1*) into buffer *B1*. *B1* then fills and is assigned to another thread to be flushed. *t1* then inserts pair (*k*, *v2*) into buffer *B2*. *B2* then fills and is assigned to another thread to be flushed. If the thread inserting *B1* stalls, *B2* could be flushed before *B1* meaning that it would be possible for the B-Tree to contain (*k*, *v1*)! In fact, we cannot guarentee any sort of consistency for the B-Tree at any point since buffers can be flushed in any order. Clearly this is not useable.

To fix this problem we tag each payload with a monotonically increasing version number after an insert has been placed in an insert buffer or at the time of insert if the update is performed directly on the B-Tree. We then use this version number to ensure that the most up to date value for a given key is contained in the B-Tree. Specifically, when we insert into the B-Tree, if the key already exists in the tree we check the version numbers of the payloads and store the payload with the greater version number.

```
# version is a shared atomic counter
def insert(key, payload):
    restart:

    current_buffer = get_current_buffer()
    # try locking the buffer for inserts
    if not current_buffer.is_locked_by_this_thread():
        release_shared_locks()
        if not current_bufffer.try_lock_shared():
            goto restart
    # buffer is full
    if current_buffer.insert(key, payload, &version) == FAILURE:
        release_shared_locks()
        # try to take ownership and flush
        if current_buffer.take_ownership():
            replace_insert_buffer()
            # wait for other theads to finish inserts
            current_buffer.lock_exclusive()
            for k,p in current_buffer:
                 btree.insert(k, p)
            # clear buffer, update minimum version, and unlock
            current_buffer.reset(version)
            current_buffer.unlock_exclusive()
        # tag payload
        versioned_payload = Versioned(payload, version++)
        btree.insert(key, versioned_payload)
```

Flushing Buffers

If a thread attempts to insert into a buffer and the insert fails, this means that the buffer is full. In this event the thread, first releases its shared lock on the buffer. After releasing the lock, the thread attempts to take ownership of the buffer with an atomic compare exchange operation. If the compare exchange operation fails, the thread inserts directly into the B-Tree and returns to the caller. If the compare exchange operation succeeds, the thread then replaces the buffer with a free buffer from a preallocated pool and notifies all other threads that the new buffer is ready. The thread must then wait for all other threads to complete their inserts. To do this, the current thread takes an *exclusive* lock on the buffer, waiting for other threads to release their shared locks on the buffer. Once the exclusive lock is acquired, the thread then inserts all of the buffered inserts in the B-Tree in a normal fashion. Finally, the thread clears the buffer by setting the current insert position to zero and updating the minimum version number of the buffer to the current version number.

Reading

Buffering the inserts also adds another complication to consistency. Since inserts into the B-Tree are delayed for an indeterminiate amount of time, keys may not be immediately visible in the B-Tree after the insert procedure completes. Furthermore, multiple buffers could be getting flushed at any particular time. To solve this problem we allow threads to read directly from the buffers. Before explaining the procedure we must first explain the notion of a *valid* buffer read. We say that a read is *valid* if the version number of a payload is greater than the minimum version number of the buffer and the less than or equal to the maximum version number of the read. As mentioned above, writes into the buffers are *not* atomic, this means that a thread could read an updated key with a payload from another write or read an updated payload with a stale key. To prevent this, the threads synchronize on the version number, this ensures that if a thread reads data with a version number less than or equal to the maximum version, the key and payload written to the buffer will also be visible. This ensures that torn writes

are never returned but it doesn't prevent the buffer from being recycled and overwritten while we are reading it. To prevent this after reading the payload, we compare the current minimum version number to what it was when we started reading the buffer. If the minimum version number has changed, the read maybe invalid, but the contents of the buffer have been flushed, hence the key will be found in the B-Tree.

The perform a read, we first load the current version number (without incrementing). Next we get the current insert buffer and search it for the key. If the key is found and the read is valid payload is retrived and store. Next, we iterate through the buffer pool, looking for buffers that are currently being flushed. If a buffer is currently being flushed, we search it and if the key is found, and the read is valid, the payload is retrived and stored. Finally, we search the B-Tree, if the key is found, we retrive the payload and store it. Finally, we return the payload that has the greatest version number. Searching in this order ensures that any insert, buffered or visible in the tree is always found, since any operation performed which is not found in the buffers will be visible in the B-Tree.

Experiments

Experimental Setup

All experiments where run on single machine with a AMD Ryzen Threadripper 2950X (16 cores/ 32 threads) with 64GB of quad channel running a 2933MHz. The operating system was Pop_OS 18.04 with linux kernel version 5.0.0. All code with compiled with g++11, with -O3, -ltcmalloc_minimal, -lpthread, and OpenMP. For each test a worker thread would fetch and operation to do, perform the operation and then fetch the next operation to be done. That is, the operations a particular thread does during the experiment are not determined a priori. This explains the drastic difference of our experimental results when compared with [3], where the authors interleaved the operations by assigning each thread an equal portion of operations a priori in a round-robin fashion (e.g. if there are 2 threads, each thread is assigned every other operation before workload execution begins). We dynamically assign operations for two reasons. First, it is a much more realistic scenario, if the operations were know a priori the tree could be constructed in a much smarter way than sequentially executing the operations. Second, when the operations are assigned a priori for sequential workloads, some worker threads end up falling behind the others which reduces contention. The reduction in contention increases throughput as the number of threads increases, essentially having stragglers benefits overall throughput. Such an effect hides the effectiveness (or lack there of) of contention regulating mechanisms for our problem setting.

Workloads

We evaluate our solution of four workloads. Each workload has 8 byte integer keys, and 8 byte integer payloads. We have two insert only work loads and two insert and read workloads. Each has a random and sequential variants. Both insert only workloads are 50M inserts, while the insert and read workloads are 50M inserts and 10M reads. For the insert and read workloads, the reads a sequenced immediately after the insert of the same key, making it very likely that the read will conflict with the write. We decided to do this to further stress the contention regulation mechanisms of the algorithms tested.

Results

Discusssion

Sources

[1] Leis, Viktor et al. “Optimistic Lock Coupling: A Scalable and Efficient General-Purpose Synchronization Method.” IEEE Data Eng. Bull. 42 (2019): 73-84.

[2] J. J. Levandoski, D. B. Lomet and S. Sengupta, "The Bw-Tree: A B-tree for new hardware platforms," 2013 IEEE 29th International Conference on Data Engineering (ICDE), 2013, pp. 302-313, doi: 10.1109/ICDE.2013.6544834.

[3] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G. Andersen. 2018. Building a Bw-Tree Takes More Than Just Buzz Words. In *Proceedings of the 2018 International Conference on Management of Data* (*SIGMOD '18*). Association for Computing Machinery, New York, NY, USA, 473–488. DOI:https://doi.org/10.1145/3183713.3196895

[4] Goetz Graefe. 2011. Modern B-Tree Techniques. *Found. Trends databases* 3, 4 (April 2011), 203–402. DOI:https://doi.org/10.1561/1900000028

[5] Philip L. Lehman and s. Bing Yao. 1981. Efficient locking for concurrent operations on B-trees. *ACM Trans. Database Syst.* 6, 4 (Dec. 1981), 650–670. DOI:https://doi.org/10.1145/319628.319663